

UDC 004.942, 519.876.5

Doi: 10.31772/2712-8970-2025-26-1-34-47

Для цитирования: Лелеков А. Т. Динамическое моделирование технических систем на Python // Сибирский аэрокосмический журнал. 2025. Т. 26, № 1. С. 34–47. Doi: 10.31772/2712-8970-2025-26-1-34-47.

For citation: Lelekov A. T. [Technical system simulation with Python]. *Siberian Aerospace Journal*. 2025, Vol. 26, No. 1, P. 34–47. Doi: 10.31772/2712-8970-2025-26-1-34-47.

Динамическое моделирование технических систем на Python

А. Т. Лелеков

Федеральный исследовательский центр «Красноярский научный центр
Сибирского отделения Российской академии наук»
Российская Федерация, 660036, г. Красноярск, Академгородок, 50
E-mail: a.t.lelekov@yandex.ru

Аннотация. Представлены результаты разработки диспетчера для совместного выполнения имитационных моделей многокомпонентных систем. Программное обеспечение реализовано на Python, что обеспечивает интеграцию множества библиотек для управления и анализа данных. Обмен данными осуществляется через UDP-пакеты, поддерживающие разные языки программирования. Это упрощает реализацию технологии hardware-in-the-loop, улучшая разработку систем управления. Пример использования диспетчера представлен на модели системы ориентации космического аппарата CubeSAT с магнитной системой ориентации. Приведен алгоритм B-Dot и результаты моделирования переходного процесса. Исходный код доступен под лицензией BSD на GitFlic, а документация – на ReadTheDocs.

Ключевые слова: система ориентации и стабилизации, магнитная система, CubeSAT, имитационное моделирование, совместное решение.

Technical system simulation with Python

A. T. Lelekov

Federal Research Center “Krasnoyarsk Science Center of the Siberian Branch
of the Russian Academy of Sciences”
50, Akademgorodok, Krasnoyarsk, 660036, Russian Federation
E-mail: a.t.lelekov@yandex.ru

Abstract. The results of the development of a scheduler for the joint execution of simulation models of multicomponent systems are presented. The software is implemented in Python, which allows integration with numerous libraries for control and data analysis. Data exchange is carried out via UDP packets that support different programming languages. This simplifies the implementation of hardware-in-the-loop technology, improving the development of control systems. An example of using the scheduler is demonstrated on the model of the attitude determination and control system of a CubeSat spacecraft with a magnetic orientation system. The B-Dot algorithm and the results of simulating the transient process are provided. The source code is available under the BSD license on GitFlic, and the documentation is available on ReadTheDocs.

Keywords: orientation and stabilization system, magnetic system, CubeSAT, simulation modeling, joint solution.

Introduction

As part of the implementation of the project "Space mission ReshUCube-3", the laboratory of small spacecraft (SSC) of Reshetnev University together with the laboratory of Space systems and technologies of the FRC KSC SB RAS is developing an orientation system for a small satellite of the CubeSat 1U format. It has an active magnetic orientation system, the structure of which is shown in Fig. 1. It is a classical system with feedback.

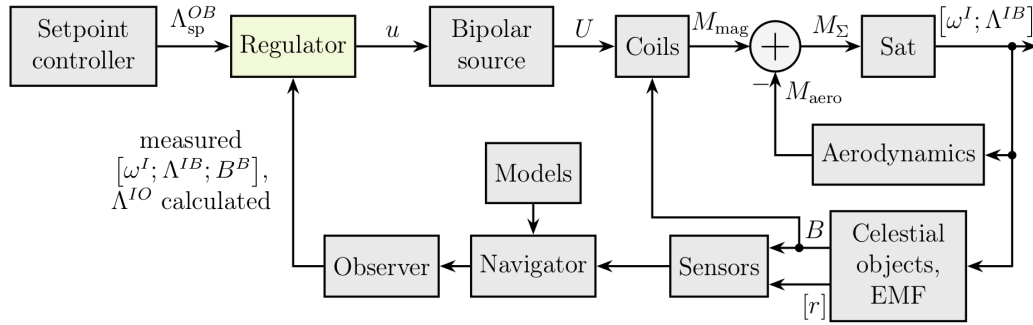


Рис. 1. Структурная схема системы ориентации КА ReshUCube-3

Fig. 1. ADCS of ReshUCube-3 structure

The state of the spacecraft (SC) is described by the angular velocity ω^I and the rotation quaternion Λ^{IB} of the associated coordinate system (superscript B, body axes) relative to the inertial space (I, inertial). The state depends on the applied rotational moment, which is composed of the moment from the M_{aero} aerodynamic forces and the magnetic moment M_{mag} . The controlling moment is the magnetic moment, which is created by the interaction of the magnetic fields of the electromagnetic device (EMD, magnetic coil) and the vector B of the Earth's magnetic field (EMF).

Several coordinate systems are used to control the orientation - some vectors are simply more convenient to specify in their own coordinate system. Firstly, this is a bound coordinate system (body axes, index B), the axes of which are rigidly connected to the apparatus and rotate together with it. They are usually located along the construction axes of the spacecraft. Secondly, the inertial coordinate system (inertial, index I), which is tied to the inertial space, to the direction to the stars. It does not rotate. Thirdly, the orbital coordinate system (orbital, index O), always directed by the x_O axis to the center of the Earth (along the local vertical), and the x_O axis is directed along the spacecraft velocity vector. This system is convenient for determining the direction to the Earth and taking into account aerodynamic forces and moments. The superscripts of a vector or quaternion indicate in which of the coordinate systems its components are taken.

The control device generates a control signal in such a way that the measured rotation quaternion Λ_{meas}^{IB} becomes equal to the desired quaternion Λ_{des}^{IB} , i.e., so that the spacecraft takes the specified orientation. In form, this is a classic PD controller for such tasks (for the D-channel, the angular velocity ω_{meas}^I is used). An excellent review of the current state of magnetic orientation systems is given in [1].

In the given control system, the navigation algorithm and the observer algorithm are quite complex, which, based on sensor readings, calculate the most statistically reliable values of the spacecraft state variables.

In the structural diagram, it is possible to distinguish groups of blocks that have different natures. For example, spacecraft rotations, aerodynamic and magnetic moments have a physical nature and act continuously; the controller, observer and navigator are algorithms that operate in the onboard computer with some periodicity. It is clear that in simulation modeling, these groups must be calculated using different computational methods, linking them into a single calculation in order to obtain a joint solution.

This calculation method is called co-simulation, the essence of which is as follows. Simulation models of parts of the system are combined with their calculation method (solver) into calculation modules. The modules are sequentially, in the order specified by the system structure, launched for calculation at a certain step of model time. The calculation results are transmitted as initial conditions for the modules following in the calculation order. Thus, the modules are interconnected, and the overall solution is joint.

Problem Status Overview

A good and very detailed overview of the current state of this technology is given in [2]. Inter-modular interaction is provided by an orchestrator - a module manager that sets the model time, determines the order of launching modules for calculation and ensures the transfer of data between modules, from the output to the inputs of the next one.

ALGORITHM 3: Generic Jacobi-based orchestrator for autonomous CT co-simulation scenarios.

Data: An autonomous scenario $cs = \langle \emptyset, Y_{cs}, D = \{1, \dots, n\}, \{S_i\}, L, \emptyset \rangle$, and a communication step size H .

Result: A co-simulation trace.

$t := 0;$

$x_i := x_i(0)$ for $i = 1, \dots, n;$

while true do

 Solve the following system for the unknowns:

$$\begin{cases} y_1 = \lambda_1(t, x_1, u_1) \\ \dots \\ y_n = \lambda_n(t, x_n, u_n) \\ L(y_1, \dots, y_n, y_{cs}, u_1, \dots, u_n) = \bar{0} \end{cases} \quad (15)$$

$x_i := \delta_i(t, x_i, u_i)$, for $i = 1, \dots, n;$

 // Instruct each SU to advance

$t := t + H;$

 // Advance time

end

Рис. 2. Алгоритм оркестратора для систем с непрерывным временем [2]

Fig. 2. Orchestrator algorithm for continuous-time systens [2]

Fig. 2 shows a generalized orchestrator algorithm for continuous-time systems. It explicitly identifies a cycle over model time, in which:

1. Model outputs are calculated taking into account the imposed constraints. Here, a system of non-linear equations is solved that calculates the operating point of the system.
2. Model state variables for the next step are calculated.
3. Step H is taken over time.

The cycle is performed until the final time is reached.

In the given algorithm, $\lambda(t, x, u)$ is the function for calculating the model output y ; x and u are the state variables and inputs of the model; δ is the function for calculating the system state variables; H is the time step size; L is the functional that specifies the constraints.

The authors of [2] distinguish discrete, continuous and hybrid models depending on the type of interaction of models in time. They differ in the features of orchestration during joint calculation. The review considers in detail the problems of joint modeling specific to each type of system, for example, the use of constraints imposed on solutions, algebraic cycles, strategies for initializing models at the operating point, control of convergence and stability of the solution, accuracy and validity of coupled models.

In practice, inter-model interaction is implemented in different ways. A common pattern is the "Publisher-Subscriber" message transfer pattern [3], in which a message (e.g., a calculation result) is placed by the publisher in a data transfer channel or database, and the subscriber selects only the messages it specifically requires.

In some implementations, the publisher sends messages to an intermediary (broker). In this case, subscribers must register a subscription with the broker, which stores and forwards messages to the subscriber. Subscribers can subscribe to specific messages at the coding stage, during application initialization, or during execution.

For example, the well-documented HELICS (Hierarchical Engine for Large-scale Infrastructure Co-Simulation) framework [4] uses a broker that organizes inter-model communications (provides information about model interfaces), and message exchange occurs at the horizontal model-to-model level, bypassing the broker (Fig. 3). Subscription uses JSON configuration files that are read during initialization. In addition, in HELICS, the broker functions as an orchestrator, synchronizing message exchange.

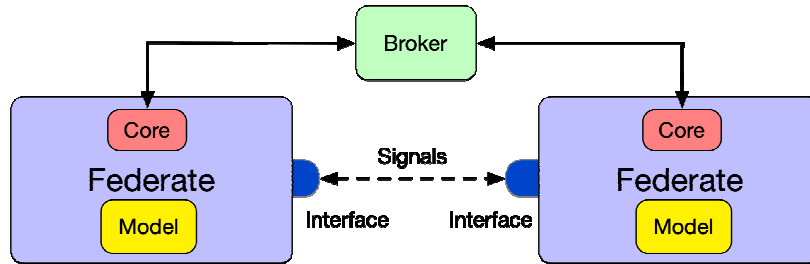


Рис. 3. Базовая архитектура фреймворка HELICS [5]. Здесь Federate – вычислительный модуль с ядром Core (управление моделированием и интерфейс сообщений) и выполняемым кодом модели

Fig. 3. Base architecture of HELICS framework [5]. Here Federate is a computational module containing a core (simulation control and message interface) and executable model code

The architecture of the MOSAIC package, written entirely in Python, includes a core and a set of interfaces (Fig. 4). The core, in which the orchestrator is implemented, consists of two components: a scheduler and a model coordinator.

The user writes a script that sets the modeling mode. Based on the script, the scheduler determines the parameters of the models, ensures the sequence of execution and synchronization of the models, and acts as a broker for data exchange. The coordinator is responsible for communication between models, using interfaces for specific protocols for inter-model data exchange.

In the domestic simulation modeling complex SimInTech, a broker is not used: the model does not register signals, but simply takes them by requests to the common intermodel signal database. The architecture of the complex is shown in Fig. 5.

The SimInTech complex is a system with model compilation into machine code, which allows for very fast simulation. To set up custom control algorithms containing complex operations for processing sensor signals and calculating control, a built-in programming language is used. It is a dialect of the Pascal language adapted for the tasks of programming simulation models operating in time. For example, it includes code sections that are executed at individual stages of simulation (compilation, initialization, finalization), there are capabilities for accessing system functions and variables, as well as for working with a model signal database, etc.

The language is quite deeply specialized and is not fully compatible with the Pascal language (for example, there is no way to set complex types of variables and objects), there are no third-party compilers and debuggers for it.

The latest versions of SimInTech have integration with the Python programming language - you can execute a script once (for example, during initialization). Since Python code is not compiled, it will take quite a long time to execute when executed cyclically (starting the interpreter, passing the script and parameters, execution, passing the results).

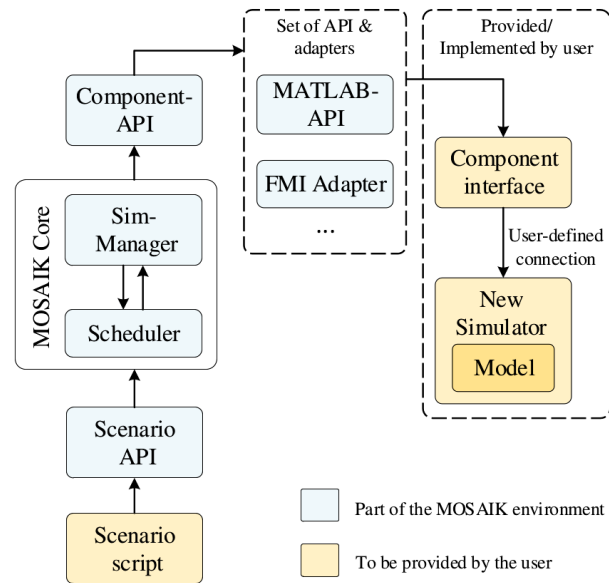


Рис. 4. Архитектура фреймворка MOSAIC [6]

Fig. 4. Architecture of MOSAIC framework [6]

It is executed by a Python interpreter external to SimInTech, and with this method of launching it is extremely difficult to implement storage of the state of objects (this was required for the SGP4 algorithm, which determines the current position of the spacecraft [7]).

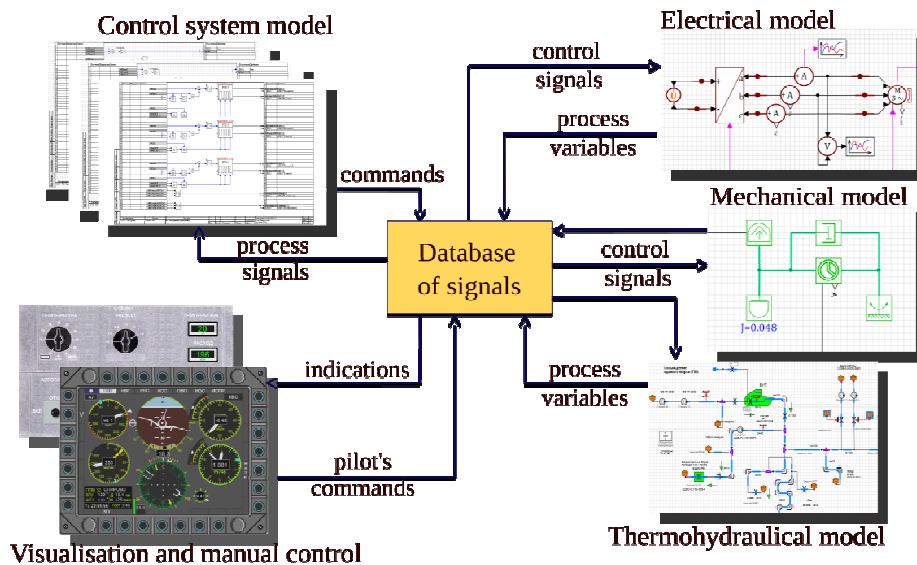


Рис. 5. Архитектура комплекса имитационного моделирования SimInTech [8]

Fig. 5. Architecture of SimInTech dynamic simulation environment [8]

SimInTech allows you to develop a block with your own algorithms using external SimInTech tools (for example, in the form of a dynamic .dll library), but the creation procedure is described only for Windows, using the Delphi environment. In 2020, the SimInTech system was successfully ported to Linux OS [9], but, unfortunately, to date there is no description of the procedure for creating and connecting blocks in the form of dynamic .so libraries. Debugging in the SimInTech programming environment is not very convenient, although all the basic tools are available (breakpoints, viewing the state of variables, etc.). The static code analyzer (linter) is not implemented. As a result, a number of

features of the software discussed above did not allow the implementation of the orientation system model in it; the code was quite cumbersome, fast-growing and difficult to debug.

Therefore, it was decided to rewrite all algorithms and models in Python, for the joint execution of which a dispatcher was written for the joint execution of models calculated by different algorithms.

Considering modeling systems implemented using Python, we should mention the excellent and well-developed Basilisk system [10], developed at the Astrodynamics Research Center of the University of Colorado. This compiled system is written in C++, and the Python code is linked (and compiled) to the system via the Software Interface Generator (SWIG). Accordingly, the models are executed quite quickly. Basilisk was written specifically for modeling space systems and has a very good graphics subsystem. It also has the ability to connect C++ code, implement hardware-in-the-loop technologies, as well as software-in-the-loop programs. Like all programs with extensive functionality, well-developed and with a long history, Basilisk is quite difficult to use, but this complexity is fully justified. Frankly speaking, if at the time when the author started working on the orientation system, the Basilisk system had been encountered, then it is unlikely that the development of his own modeling system would have begun.

The orientation system model presented in the report [11] is written entirely in Python. Its peculiarity is that it uses a simplified technology of joint modeling: a common matrix of equations is formed, solved by the LSODA algorithm for continuous systems per time step; then the orientation system algorithms are executed. The system has algorithms and equations of individual blocks, each of which ensures the formation of its part of the equation system.

Architecture

Figure 6 shows the architecture of a dispatcher for simulation modeling of control systems.

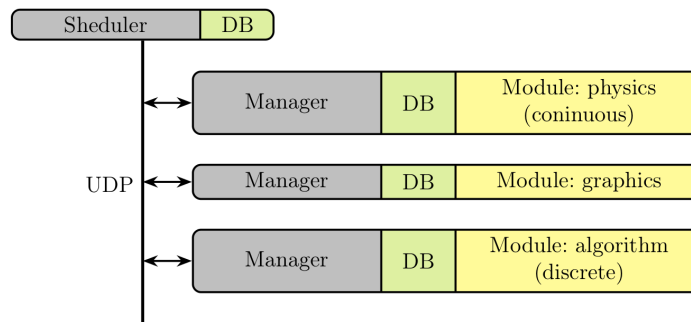


Рис. 6. Архитектура диспетчера имитационного моделирования pySimScheduler [12]

Fig. 6. Architecture of dynamic simulation model manager pySimScheduler [12]

Each model is implemented as an independent calculation module. It is launched in a separate process and contains both the equations and algorithms of the model, and the methods of numerical calculation of these algorithms. The module must calculate its output values for a certain time interval.

Similar to how it is implemented in SimInTech, the module can be in several states (Fig. 7). The transition between these states occurs according to the dispatcher commands or in response to the module's internal events.

After receiving the command to start the simulation, the module switches (A) from the Idle waiting state to the Run working state; in this case, the graphics and internal variables of the module are initialized (the initial state is set). After this, the dispatcher cyclically issues a command to each calculation module to perform a calculation step (B), passing the initial data (the calculation result of the modules adjacent to it) to the module and taking the calculation results (to pass them on to subsequent modules). At the end of the calculation time, the module switches to the waiting state (C), performing actions to finalize the calculation (finishing the graphics, closing files). If an internal error occurs during the calculation, the module switches to the error state (E) with the error message to the dispatcher,

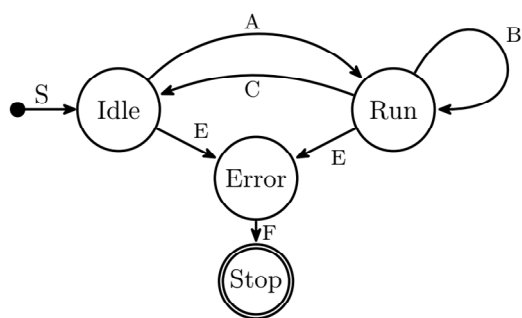


Рис. 7. Машина состояний системы
pySimScheduler

Fig. 7. State machine of pySimScheduler
model manager

and then stops. In this case, on the command of the dispatcher, the remaining modules switch to the waiting state.

Data exchange is performed via UDP packets, which contain variables from the common database required for a specific module. Each module contains a database manager. It unpacks/packs packets and distributes data according to the internal variables of the module class. The system is based on [13], where exchange is performed with the equipment of the stand, which works like a calculation module.

The database must contain the actual database of variables that will be common between modules. It also contains a list of modules indicating which vari-

ables participate in the exchange (which are transmitted, and which are received).

After the module is initialized, a Manager object will appear in its attributes, which is the database manager. It will create (during its initialization) the required attributes in the module object, which are fields of the common database – they will be synchronized at each calculation step.

The general database, its fields, and the composition of specific variables for exchange with specific modules are set in the container class in the form of attributes:

Listing 1. Sample database code

```

class DataBase():
    # --- common variables ---
    t = 0. # model time
    dt = 0.25 # calculation step
    tmax = 4. # simulation time
    cmd = 0 # command to all modules

    # --- user variables ---
    U = np.array([0.1, 0., 0.]) # control
    L_B = np.zeros(3) # magnetic field of coils
    B_I = np.array([0., 1e-5, 0.]) # magnetic field (I), [T]
    Mm_B = np.zeros(3) # torque from coils
    w_B = np.zeros(3) # angular velocity
    q_IB = quat.Quaternion() # quaternion of the bound SC relative to the inertial
  
```

Modules are called in the order they appear in the Tasks dictionary. For example, Listing 2 specifies three modules (Rotation, Control, and Plot2D), whose names are keys in this dictionary:

Listing 2. Example of defining module call parameters

```

Tasks = {
    'Rotation': {'Keys': 't,cmd,dt,
                        U,L_B,B_I,Mm_B,
                        w_B,q_IB'},
    'Control': {'Keys': 't,cmd,U,B_I,q_IB'},
    'Plot2D': {'Keys': 't,cmd,dt,U,w_B,q_IB',
               'Addr': ('188.162.92.100', 6502),
               'noAnswer': True}
  
```

The values corresponding to the keys are also dictionaries (dict). They have a mandatory Keys, which specifies the database fields that will be exchanged with this module at the specified address.

If data is exchanged with a remote system, then it is necessary to specify the network address to which data packets will be sent. In this case, the Addr key is added, the value of which must be a tuple and have the form (ip address, port).

It is possible to configure the exchange so that the module does not return data, but only receives them. This is convenient when the module solves computationally difficult problems of visualizing two-dimensional and three-dimensional graphs. In order not to wait for an answer, the noAnswer flag is used.

Since standard methods for solving differential and other equations are used to obtain a joint solution in each of the models, and data exchange is completely predictable, the joint solution will be adequate and accurate. It is clear that for a stable solution, the question of choosing the step length of variable exchange arises. It is considered in many publications (see review [2]), and with a reasonable approach to choosing the parameters of the solution methods, the joint solution will be adequate.

Writing and executing models

The model code looks like a description of a Python class, which must have methods with specific names that are called during transitions between states. It is necessary to specify the calculation code of the model in them. The state machine (see Fig. 7) and the database manager are implemented in the parent class, from which the class with the model description must be inherited.

From the code, it is clear that it is necessary to override (if necessary) the following methods, specifying the required model logic in them:

- Setup – is run once when an instance of the model class is created;
- Initialize – is executed once during model initialization before running the calculation;
- Run – model calculation. It is cyclically for each time interval dt;
- Finalize – is executed once, at the end of the calculation.

On transitions to the Error and Stop states, the module's internal service methods are executed, into which the user should not enter the model logic.

At the end of the file with the model code, you need to create an instance of the class and transfer control to the internal state machine:

Listing 3. Example of transferring control to a module

```
model = Controller(TaskList=db.Tasks, DB=db.DataBase(), isSheduler=False, isRealTime=False)
model.Manager.Loop()
```

The parameters of the model class are specified:

- TaskList: list of modules Tasks;
- DB: shared database object;
- isSheduler: a flag indicating whether the module contains a dispatcher and runs other modules.

One of the modules must be the master module, run all the others and ensure data exchange. It must have the isSheduler=True flag and be run last. It is convenient to make the module whose code we are currently working with the master module;

- isRealTime: flag indicating whether we are working in real time (the t attribute corresponds to real time). This mode is usually used when exchanging data with real equipment.

Each module must be run in a separate Python interpreter, with the console displaying the module's state. After the master module has been launched, the system begins to simulate - it runs the state machine. After the simulation time has expired, the slave modules go into the Idle state and wait for further commands. When restarting, the model uses the initial conditions specified in the database at the time of Setup; initial conditions can be specified directly in the model code.

Application

To determine the quaternion of the spacecraft rotation relative to the inertial space, the navigator algorithm is used.

Let a set of unit “measured” vectors b be received from the sensors in the current angular and spatial position of the spacecraft, presented in a related coordinate system. For a small spacecraft with a limited set of sensors, this could be, for example, the direction to the Sun, Earth, and the EMF vector.

Also, for the current spatial position of the spacecraft, the same vectors are calculated – a set of “standard” (reference) unit vectors r . They are specified in the inertial reference system.

It is necessary to find a rotation that will combine the reference vectors with the measured vectors with the smallest error. This rotation will determine the angular position of the associated coordinate system relative to the inertial one.

The calculation of reference vectors must be performed on board. The initial data for this algorithm are the spacecraft coordinates in the inertial coordinate system, which are calculated based on the TLE parameters using the SGP4 algorithm. This is a fairly complex algorithm (Fig. 8), proposed in [14] and translated into many programming languages. To determine the coordinates, the TLE parameters are transmitted to the spacecraft from time to time (at intervals of approximately one week). The SGP4 algorithm is quite resource-intensive, and therefore the exact calculation is performed periodically (daily), and between its calls an approximate, extrapolated solution is calculated.

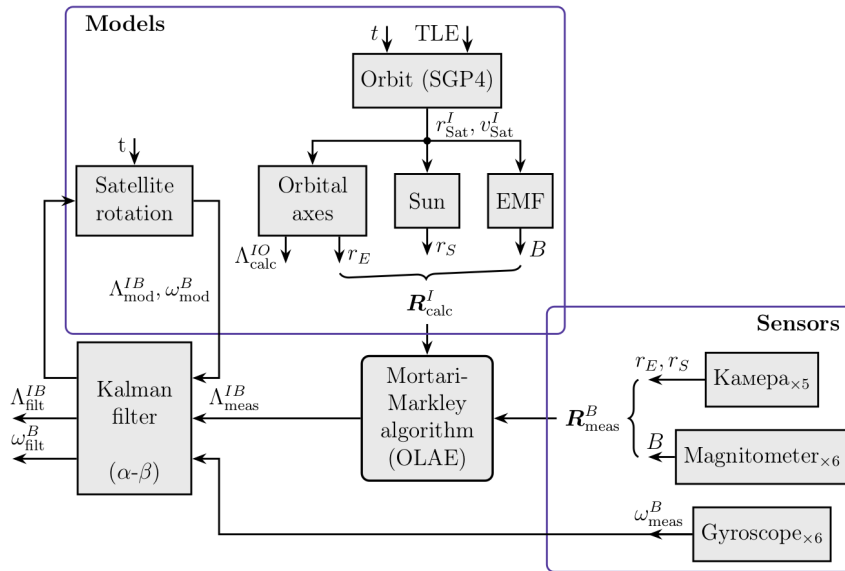


Рис. 8. Структура алгоритма навигатора

Fig. 8. Navigation algorithm structure

In the simulation model of the system, calculation of reference vectors is also required for sensor models. In this case, the algorithm models the surrounding world – the “real” position of the Sun and the vector of the EMF, calculating the corresponding vectors r^* in the inertial system. Based on these vectors, the sensor model forms the measured vectors b , adding noise and bias.

If the vectors r and r^* are calculated using the same algorithm, they will differ only in the added noise. Therefore, to increase the robustness of the system, it was decided to use different implementations of the algorithms: in the onboard software, to use the C/C++ code from [7; 15], and in the sensor model, to use the OreKIT astrodynamics library [14].

A simplified manager was developed to link the C++ code with the dispatcher, which is written in Python. It implements a similar state machine (see Fig. 5), calling the necessary procedures for initialization (required for the SGP4 algorithm), calculation per time step, and finalization of the model. Since the format of structures in Python and C++ is different, the cppstruct library [17] is used to work with UDP packets on the model side. It allows unpacking and packing UDP packets into a format accepted by the struct library (an example of the code is given in the ex5.1 directory of the repository [13]). At the moment, the manager code is formed by calling a special dispatcher method; the

code will form a C++ structure with field names corresponding to variables in the database. The programmer must independently set calls to the necessary procedures, passing them data from this structure and filling its fields after the calculation.

Control of the spacecraft's rotational motion

Figure 9 shows the structural diagram of the rotational motion model. It details the EMU, KA and Aerodynamics blocks shown in Fig. 1.

Control with aerodynamic feedback is quite complex. A detailed examination of the control principle in this case requires a separate article. Briefly, it can be noted in advance that the aerodynamic moments have a value of about 10 % of the moments created by the magnetic orientation system. Therefore, in the first approximation, we will not take this feedback into account.

The rotational motion of a rigid body is described by two equations – dynamics and kinematics [16]. The first, the Euler equation, written in a bound coordinate system (CS) in matrix form, has the form

$$J\dot{\omega}^B + \omega^B \times (J\omega^B) = M^B,$$

expressing from which the derivative of the angular velocity, we have

$$\dot{\omega}^B = J^{-1} (M^B - \omega^B \times (J\omega^B)),$$

where J is moment of inertia tensor; ω^B – angular velocity vector; M^B is torque moment.

The equation of rotational kinematics, written in quaternion form, has the form

$$\dot{\Lambda}^{IB} = \frac{1}{2} \Lambda^{IB} \circ \omega^B,$$

where Λ^{IB} is quaternion of rotation of the associated SC relative to the inertial space.

The moment acting on the spacecraft consists of aerodynamic and magnetic moments. The magnetic moment is used to control the motion. It is created by the interaction of the field of the coils L and the Earth's magnetic field B . Like the needle of a magnetic compass, the system of coils rotates until the vector L coincides with the Earth's field B . The rotational moment can be calculated as $M = L \times B$.

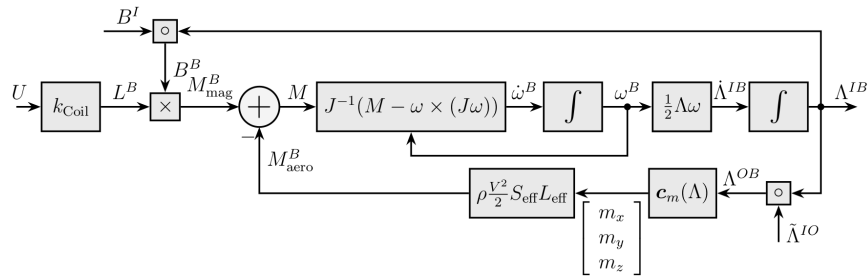


Рис. 9. Модель вращательного движения КА

Fig. 9. Model of satellite rotational movement

For simplicity, we assume that the main feedback (including the navigation and observer algorithms) works perfectly, its coefficient is equal to one. Therefore, we close our system and consider the simplest algorithm, which is used on all (with very rare exceptions) low-flying small spacecraft - the B-Dot angular velocity damping algorithm. It is used at the initial stage of the mission to dampen the rotation that inevitably occurs after the small spacecraft is pushed out of the launch vehicle's release container.

The classic B-dot algorithm is [21]

$$M = -k \frac{\dot{B}}{\|B\|},$$

where k is a coefficient, scalar or vector (for the case, for example, of a large difference in the moments of inertia along the axes of the spacecraft). The choice of the coefficient is a separate interesting problem (see, for example, [18]), but even manual selection gives acceptable control quality. Normalization of the magnetic field derivative vector (denominator) increases the stability of the coefficient in the control loop, although magnetometer noise, of course, remains.

In code, the B-dot algorithm looks simple:

Listing 4. A variant of the B-dot algorithm implementation

```
kBDot = 500
def BDot(self):
    """ B-Dot algorithm on normalized vector of Earth's magnetic field """
    dB = self.B_Bn - self.Buf.B_Bn[-1]
    dt = self.t - self.Buf.t[-1]
    return -self.kBDot * dB/dt
```

Here, the Buf object is a buffer that stores the values of the variables t and B from previous iterations. They are updated in the main algorithm (see examples in the project's GitFlic repository [19]).

Fig. 10 shows the transient process for the angular velocity of the spacecraft – the component of the angular velocity ω^B and the vector of the measured magnetic field B^B , in the axes of the associated coordinate system. It is clear how the system cannot damp the component ω_X – it coincided in direction with the vector, and the magnetic system physically cannot create a moment around the vector of the magnetic field.

During rotation, the most energy-consuming axis of the spacecraft is the one with the largest moment of inertia. When the algorithm is running, when a counteracting moment is created, this component is the most difficult to extinguish. Therefore, the spacecraft almost always turns so that this axis becomes collinear to the magnetic field vector B , which is observed in the graphs of the transition process for the components of the angular velocity.

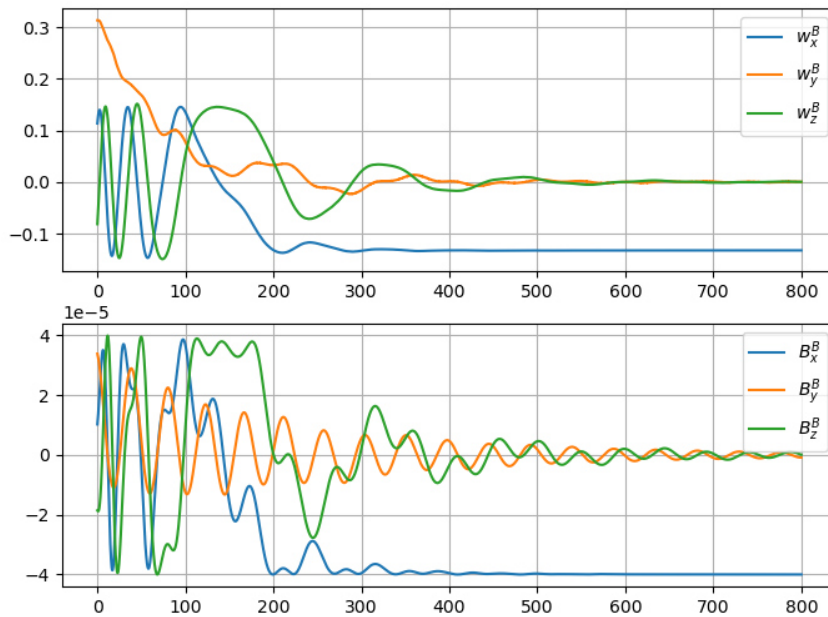


Рис. 10. Переходной процесс при гашении угловой скорости алгоритмом B-Dot

Fig. 10. Transient of B-Dot detumbling algorithm

Conclusion

The paper presents the results of developing a dispatcher designed for joint execution of simulation models that make up a multicomponent system. As a result of modeling, a joint solution to the transient process is obtained, which allows for complex testing of algorithms and models. This helps to identify errors that may be difficult to detect during isolated testing of individual subsystems.

A feature of the developed software is its implementation in the popular Python programming language, which allows for seamless integration of a huge number of libraries for Python, including for the development of control systems and data analysis.

Data exchange between models is performed via UDP packets, which allows using not only Python, but also other programming languages for writing models. For the same reason, it is quite simple to implement the hardware-in-loop technology (a corresponding example is given in the article [11]), which allows for more efficient development of control systems.

An example of using the developed manager to create a model of the orientation system of a small CubeSAT spacecraft with a magnetic orientation system is presented; an example of implementing the B-Dot algorithm and the results of modeling the transient process are given.

The developed system is used for simulation modeling and development of the orientation system of the ReshUCube-3 Space Mission project.

Links to the source code of the system (BSD License) are provided, which is posted in the public domain on GitFlic [21], the documentation is posted on the ReadTheDocs platform [10].

Библиографические ссылки

1. Ovchinnikov M. Yu., Roldugin D. S. A survey on active magnetic attitude control algorithms for small satellites // Progress in Aerospace Sciences. 2019. Vol. 109. P. 100546.
2. Co-Simulation: A Survey / Gomes Cláudio, Thule Casper, Broman David, Larsen Peter Gorm, Vangheluwe Hans // ACM Computing Surveys. 2018. Vol. 51, No. 3. P. 1–33.11.
3. Hohpe G., Woolf B. Enterprise Integration Patterns Designing, Building, and Deploying Messaging Solutions. Pearson Education, Limited, 2012. P. 736.
4. HELICS: A Co-Simulation Framework for Scalable Multi-Domain Modeling and Analysis / Hardy Trevor D., Palmintier Bryan, Top Philip L., Krishnamurthy Dheepak, Fuller Jason C. // IEEE Access. 2024. Vol. 12. P. 24325–24347.
5. HELICS User Guide. [Электронный ресурс]. URL: https://docs.helics.org/en/main/user-guide/fundamental_topics/helics_terminology.html (дата обращения: 10.01.2025).
6. MOSAIK and FMI-Based Co-Simulation Applied to Transient Stability Analysis of Grid-Forming Converter Modulated Wind Power Plants / Farrokhseresht Nakisa, van der Meer Arjen A., Rueda Torres José, van der Meijden Mart A. M. M. // Applied Sciences. 2021. Vol. 11, No. 5. P. 2410.
7. Vallado David A. Fundamentals of astrodynamics and applications / ed. by McClain Wayne D. Space technology library no. 21. 3. ed., 1. printing ed. Hawthorne, Calif. u.a.: Microcosm Press u.a., 2007. 1055 p.
8. Справочная система SimInTech [Электронный ресурс]. URL: <https://help.simintech.ru>. (дата обращения: 10.01.2025).
9. Тимофеев К. А. Особенности портирования сложного модульного ПО написанного на Delphi под ОС Linux. [Электронный ресурс]. URL: <https://habr.com/ru/articles/534466/> (дата обращения: 10.01.2025).
10. Kenneally P., Piggott S., Schaub H. Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework // Journal of Aerospace Information Systems. 2020. Vol. 17, No. 9. P. 496–507.2020-09.
11. Martin G. Technical Report of ASE 372K Project. The University of Texas at Austin, USA, 2018. [Электронный ресурс]. URL: <https://github.com/gavincmartin/adcs-simulation> (дата обращения: 10.01.2025).
12. Документация к пакету pySimSheduler [Электронный ресурс]. URL: <https://pysimscheduler.readthedocs.io/> (дата обращения: 10.01.2025).
13. Lelekov A. T., Kureshov V. A. Remote Laboratory for the Design of Attitude Control Systems for Small Satellites // Instruments and Experimental Techniques. 2022. Vol. 65, No. 5. P. 858–863.

14. Revisiting Spacetrack Report #3 / Vallado David A., Crawford Paul, Hujsak Richard, Kelso T. S. // AIAA Astrodynamics Specialists Conference and Exhibit. August 2006.
15. Meeus Jean. Astronomical formulae for calculators. // 4. ed. enlarged rev. ed. Richmond, Va. : Willmann-Bell, 1988. 218 p.
16. Maisonobe L., Pommier V., Parraud P. Orekit: an Open-source Library for Operational Flight Dynamics Applications // Presented at the 4th ICATT International Conference on Astro-dynamics Tools and Techniques. Madrid, Spain. 2010.
17. Cppstruct source code [Электронный ресурс]. URL: <https://github.com/karkason/cppstruct> (дата обращения: 10.01.2025).
18. Spacecraft Dynamics and Control: The Embedded Model Control Approach / Canuto Enrico, Novara Carlo, Massotti Luca, Carlucci Donato, Montenegro Carlos Perez. Elsevier, 2018.
19. Desouky Mohammed A. A., Abdelkhalik Ossama. A new variant of the B-dot control for spacecraft magnetic detumbling // Acta Astronautica. 2020. Vol. 171. P. 14–22.
20. Avanzini Giulio, Giulietti Fabrizio. Magnetic Detumbling of a Rigid Spacecraft // Journal of Guidance, Control, and Dynamics. 2012. Vol. 35, No. 4. P. 1326–1334.
21. PySimScheduler source code [Электронный ресурс]. URL: <https://gitflic.ru/project/alexlekov/pysimscheduler> (дата обращения: 10.01.2025).

References

1. Ovchinnikov M. Yu., Roldugin D. S. A survey on active magnetic attitude control algorithms for small satellites. *Progress in Aerospace Sciences*. 2019, Vol. 109, P. 100546.
2. Gomes Cláudio, Thule Casper, Broman David, Larsen Peter Gorm, Vangheluwe Hans. Co-Simulation: A Survey. *ACM Computing Surveys*. 2018, Vol. 51, No. 3, P. 1–33.11.
3. Hohpe Gregor, Woolf Bobby. Enterprise Integration Patterns Designing, Building, and Deploying Messaging Solutions. Pearson Education, Limited, 2012. P. 736.
4. Hardy Trevor D., Palmintier Bryan, Top Philip L., Krishnamurthy Dheepak, Fuller Jason C. HELICS: A Co-Simulation Framework for Scalable Multi-Domain Modeling and Analysis. *IEEE Access*. 2024, Vol. 12, P. 24325–24347.
5. HELICS User Guide. Available at: https://docs.helics.org/en/main/user-guide/fundamental_topics/helics_terminology.html (accessed: 10.01.2025).
6. Farrokhsheresht Nakisa, van der Meer Arjen A., Rueda Torres José, and van der Meijden Mart A. M. M. MOSAIK and FMI-Based Co-Simulation Applied to Transient Stability Analysis of Grid-Forming Converter Modulated Wind Power Plants. *Applied Sciences*. 2021, Vol. 11, No. 5, P. 2410.
7. Vallado David A. Fundamentals of astrodynamics and applications. Ed. by McClain Wayne D. Space technology library No. 21. 3. ed., 1. printing ed. Hawthorne, Calif. u.a. : Microcosm Press u.a., 2007. 1055 p.
8. Справочная система SimInTech. Available at: <https://help.simintech.ru> (accessed: 10.01.2025).
9. Тимофеев К. А. Особенности портирования сложного модульного ПО написанного на Delphi под ОС Linux. Available at: <https://habr.com/ru/articles/534466/> (accessed: 10.01.2025).
10. Kenneally P., Piggott S., Schaub H. Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework. *Journal of Aerospace Information Systems*. 2020, Vol. 17, No. 9, P. 496–507.2020-09.
11. Martin G. Technical Report of ASE 372K Project. The University of Texas at Austin, USA, 2018. Available at: <https://github.com/gavincmartin/adcs-simulation> (accessed: 10.01.2025).
12. PySimScheduler documentation. Available at: <https://pysimscheduler.readthedocs.io/> (accessed: 10.01.2025).
13. Lelekov A. T., Kureshov V. A. Remote Laboratory for the Design of Attitude Control Systems for Small Satellites. *Instruments and Experimental Techniques*. 2022, Vol. 65, No. 5, P. 858–863.
14. Vallado David A., Crawford Paul, Hujsak Richard, Kelso T. S. Revisiting Spacetrack Report #3. *AIAA Astrodynamics Specialists Conference and Exhibit*. August 2006.

15. Meeus Jean. Astronomical formulae for calculators. 4 ed. enlarged rev. ed. Richmond, Va. : Willmann-Bell, 1988. 218 p.
16. Maisonobe L., Pommier V., Parraud P. Orekit: an Open-source Library for OperationalFlight Dynamics Applications. Presented at the 4th ICATT International Conference on Astro-dynamics Tools and Techniques, Madrid, Spain. 2010.
17. Cppystruct source code. Available at: <https://github.com/karkason/cppstruct> (accessed: 10.01.2025).
18. Canuto Enrico, Novara Carlo, Massotti Luca, Carlucci Donato, Montenegro Carlos Perez. Spacecraft Dynamics and Control: The Embedded Model Control Approach. Elsevier, 2018.
19. Desouky Mohammed A. A., Abdelkhalik Ossama. A new variant of the B-dot control for spacecraft magnetic detumbling. *Acta Astronautica*. 2020, Vol. 171, P. 14–22.
20. Avanzini Giulio, Giulietti Fabrizio. Magnetic Detumbling of a Rigid Spacecraft. *Journal of Guidance, Control, and Dynamics*. 2012, Vol. 35, No. 4, P. 1326–1334.
21. PySimScheduler source code. Available at: <https://gitflic.ru/project/alexlelekov/pysimsheduler> (accessed: 10.1.2025).

© Lelekov A. T., 2025

Лелеков Александр Тимофеевич – кандидат технических наук, старший научный сотрудник; Федеральный исследовательский центр «Красноярский научный центр Сибирского отделения Российской академии наук». E-mail: a.t.lelekov@yandex.ru. <https://orcid.org/0000-0003-1160-8997>.

Lelekov Alexander Timofeevich – Cand. Sc., senior researcher; Federal Research Center “Krasnoyarsk Science Center of the Siberian Branch of the Russian Academy of Sciences”. E-mail: a.t.lelekov@yandex.ru. <https://orcid.org/0000-0003-1160-8997>.

Статья поступила в редакцию 03.02.2025; принята к публикации 03.03.2025; опубликована 11.04.2025
The article was submitted 03.02.2025; accepted for publication 03.03.2025; published 11.04.2025

Статья доступна по лицензии Creative Commons Attribution 4.0
The article can be used under the Creative Commons Attribution 4.0 License